An Operating-System Infrastructure for Embedded BDI-Based Multi-agent Systems

Nilson Lazarin[1,2(✉)], Carlos Pantoja[1,3], and José Viterbo[2]

[1] Federal Center for Technological Education Celso Suckow da Fonseca (Cefet/RJ), Rio de Janeiro, Brazil
nilson.lazarin@cefet-rj.br, chon@grupo.cefet-rj.br
[2] Institute of Computing, Fluminense Federal University (UFF), Niterói, Brazil
[3] Linköping University (LiU), Linköping, Sweden

Abstract. The integration of Artificial Intelligence with Internet of Things devices has led to the emergence of Artificial Intelligence of Things, where intelligent decision-making capabilities are embedded directly in edge devices. Multi-Agent Systems (MAS), particularly those based on the Belief–Desire–Intention (BDI) model, provide a suitable paradigm for implementing autonomous behavior in such devices. However, deploying it on embedded platforms involves the computational overhead of the reasoning cycle and the complexity of the required software infrastructure. To this, this paper proposes an operating-system-level infrastructure for Embedded BDI-based MAS, introducing ACIRMa, an architectural model that extends the traditional embedded MAS architecture. Also, the paper presents an instantiation of the model through chonOS, a GNU/Linux-based operating system. Besides, an evaluation using a robotic prototype compares the specialized proposed OS against a general-purpose OS. Results show a 77.5% reduction in design time, and runtime experiments indicate improved efficiency, including a 14.4% reduction in memory consumption, a 46.8% reduction in swap usage, and a 7.2% increase in perception-processing capability.

1 Introduction

The convergence of Internet of Things (IoT) and Artificial Intelligence (AI) has led to the emergence of Artificial Intelligence of Things (AIoT), where intelligent decision-making capabilities are embedded directly into IoT devices. Since these devices generate large volumes of data, performing decision-making locally increases system autonomy while avoiding network congestion [16]. In this context, intelligent agents provide a natural approach for implementing autonomous decision-making. Agents are computational entities capable of perceiving their environment and selecting actions to achieve specific goals [39]. This paradigm has been widely adopted for developing systems capable of operating in complex environments and providing scalability, and flexibility [5].

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2027
O. Gervasi et al. (Eds.): ICCSA 2026, LNCS 16769, pp. 589–609, 2027.
https://doi.org/10.1007/978-3-032-30494-0_37

Agents often operate within Multi-Agent Systems (MAS), where multiple agents interact, exchange information, and coordinate their actions to achieve system objectives [8]. In AIoT scenarios, MAS are particularly advantageous due to their parallel and asynchronous nature, enabling distributed decision-making and the decomposition of complex problems into smaller responsibilities handled by individual agents [44]. Besides, autonomous agents typically exhibit characteristics such as *autonomy*, *reactivity*, *proactivity*, and *social ability* [8,46]. In addition, agents may exhibit *rationality* when they are expected to act in ways that promote the achievement of their goals rather than prevent them [46]. To support such capabilities, several cognitive models have been proposed for the design of intelligent agents [41].

Among these models, the Belief–Desire–Intention (BDI) architecture [7] is one of the most widely adopted approaches for constructing cognitive agents [11]. Over the past decades, the BDI model has driven the development of several programming languages, interpreters, and agent platforms [11]. One of the most influential BDI-based languages is AgentSpeak(L) [38], which provides a logical framework for programming agents using beliefs, events, and context-sensitive plans [12]. This language inspired several interpreters that bridge theoretical models of practical reasoning with executable agent systems [9].

Due to the advent of Single-Board Computers (SBC) in the 2010s, BDI-based agents can be directly situated in the physical world, perceiving and acting through sensors and actuators [21,31]. This capability enables the deployment of cognitive devices at the network edge. However, the strong abstraction provided by BDI programming, often relying on interpreters or intermediate representations, introduces challenges related to performance and resource consumption resource-constrained systems [1]. Moreover, these systems frequently depend on integration infrastructures for allow BDI-agent to interact with embedded firmware and physical devices [40].

These limitations represent open challenges for the adoption of BDI agents in AIoT environments. When successfully deployed in such contexts, BDI agents enable decentralized and autonomous decision-making directly at the network edge [26]. Providing a cognitive layer for embedded devices and mobile robots is particularly important for enabling rapid reactions to environmental changes while maintaining long-term objectives [40]. However, in the current state of the art, the computational cost of the BDI reasoning cycle and the complexity of the development and deployment process still hinder the practical adoption of embedded BDI-based MAS.

Thus, we formulate the following research problem: *in the current state of the art, the computational cost of the BDI reasoning cycle and the complexity of the development process hinder the deployment of BDI agents in embedded platforms.* To address this problem, this research investigates two questions: (i) *can the use of a dedicated operating system reduce the complexity of deploying BDI agents?* (ii) *is it possible to improve the performance of embedded BDI agents in resource-constrained platforms?*

This research is conducted following the Design Science Research (DSR) methodology [13]. To address the principle of *Problem Relevance*, Sect. 2 presents the theoretical background and a literature mapping that demonstrate the importance of the addressed problem and the lack of sufficient solutions in the current state of the art. To satisfy the *Design as a Search Process* principle, Sect. 3 discusses the rationale behind the proposed solution. Subsequently, two artifacts are introduced to address the identified problem: the ACIRMa architectural model (Sect. 3.1) and its instantiation through the *chonOS* operating system (Sect. 3.2). To satisfy the *Design Evaluation* principle, Sect. 4 demonstrates and evaluates the utility and feasibility of the developed artifacts. The implications of the results and their contributions to existing knowledge are discussed in Sect. 6, addressing the *Research Contributions* principle. Finally, the reproducibility of the research is presented in Sect. 5, ensuring the *Research Rigor* principle.

2 Background

Since the introduction of AgentSpeak(L) and its interpreters, several studies have explored the application of BDI-based agents across different domains. To understand how BDI-based embedded systems are currently being developed, we conducted a mapping study using the Snowballing method [45], as illustrated in Fig. 1. The work of Rao [38], which introduced this language, was used as the seed for the initial set of papers. In the *Forward Snowballing* step, citations of the *seed* paper were retrieved from the SCOPUS database, yielding 898 records as of October 2025. Only publications after 2010 were considered, resulting in 583 publications. To exclude works unrelated to embedded systems, we filtered titles or abstracts containing the terms "*robot**", "*IoT*", "*device*", "*physical*", "*embed**", or "*edge*", reducing the set to 155 publications. After removing duplicates, 63 candidate papers remained. Abstract reading identified 38 relevant publications presenting a *proof-of-concept* and a *model*, *approach*, *architecture*, or *tool*. After an initial full-text screening, 8 papers remained.

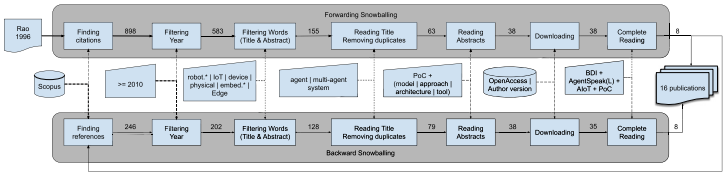


Fig. 1. Mapping review using Snowballing method.

In the *Backward Snowballing* step, references cited by these 8 papers were collected, resulting in 246 references. Applying the same year filter reduced this set to 202 publications. Filtering titles or abstracts with the same keywords reduced

the set to 128 papers. After removing duplicates and titles already considered, 79 unique papers remained. Abstract reading reduced this set to 38 candidates, of which 35 full texts were accessible. After an initial review, 8 additional papers were selected. Thus, the *snowballing* process resulted in 16 relevant papers which were analyzed in full, as summarized in Table 1.

Table 1. Publications analyzed in the literature review.

Paper	Framework	Arch/Infra	Middleware	Domain	Hardware
[20]	JaCa	CArtAgO	BrickPi	CPS	RPi 3
[27]	JaCaMo	EJasonArch	ROS	UAV	BeagleBone Black
[6]	JaCaMo	Physical Artifact	Javino	IoT	RPi Zero W
[10]	JaCaMo	Physical Artifact	Javino	IoT	RPi 3
[34]	Jason	ARGO	Javino	UGV	RPi Model B
[32]	Jason	ARGO	Javino	AmI	RPi Zero W
[19]	Jason	ARGO	Javino	UGV	RPi Zero W
[18]	Jason	Diffuser	Javino	V2V	RPi Zero W
[22]	Jason	Fuzzy-BDI	BrickPi	CPS	RPi 3
[35]	Jason	Interface Agent	RXTX	IoT	RPi Zero W
[21]	Jason	Jason Environment	BrickPi	CPS	RPi 3
[42]	Jason	Jason-ROS	ROS	UAV	RPi 3
[29]	Jason	Rason	ROS	UGV	Core i7 (laptop)
[31]	Jason	SAVI-ROS-BDI	ROS	UGV	RPi 4
[14]	Jason	SAVI-ROS-BDI	ROS	UGV	RPi 4
[43]	JasonEmbedded	ARGO	Javino	IoT	RPi Model B

The survey revealed the use of several BDI frameworks (Jason, JaCaMo, JaCa, and JasonEmbedded). All of these rely on the Multi-Agent Oriented Programming (MAOP) paradigm [4], which provides programming constructs that correspond to the abstractions used in a MAS to express social coordination.

Also, we can notice that, in general, cognitive embedded systems are composed of a *hardware platform* (a High-End IoT device) physically connected to several *hardware resources*, which may be provided by Low-End IoT devices. A Low-End IoT device is a system limited in computational resources (GPIO ports and serial connection), designed for basic sensing and actuation applications, and programmed with low-level firmware to manage sensors and actuators [30]. These devices cannot host a MAS but can be controlled by cognitive agents through serial communication protocols. Examples of such devices include microcontrollers from the ATMEGA or PIC families. A High-End IoT device, in turn, has sufficient computational resources (CPU, memory, and storage) to run a general-purpose operating system [30], hosting the MAS. These devices are single-board computers such as the Raspberry Pi or BeagleBoard.

Table 2. Embedded BDI-based systems Layers

Layer	Description
L4	*Reasoning Layer*. Hosts the cognitive agents, which receive percepts, deliberate, and send action commands.
L3	*Interfacing Layer*. Implements the communication protocol responsible for transmitting percepts and action commands.
L2	*Controlling Layer*. Resides in the microcontrollers, executing actions and translating raw sensor data into percepts interpretable by agents.
L1	*Actuating Layer*. Integrates sensors, actuators, and microcontrollers into the physical platform.

Table 3. Development Stages of embedded BDI-based systems

Step	Description
S1	*Physical Resources*. The designer selects sensors and actuators and builds the device according to the application domain.
S2	*Firmware Programming*. The designer implements functions to manage hardware resources and translate raw data into perceptions for agents.
S3	*Interfacing Programming*. The designer defines the communication protocol between the agents and the device hardware layers.
S4	*Agent Programming*. The designer implements the cognitive agents.
S5	*Services Management*. The designer configures the applications, services, and dependencies required by the embedded system OS.

Moreover, a generic architecture composed of four layers can be derived, as shown in Table 2. This four-layer view consolidates the MAOP dimensions into a practical stack for embedded autonomy: the Reasoning Layer captures AOP concerns; the Interfacing Layer materializes environment and communication abstractions and middleware; the Controlling Layer operationalizes device-level execution; and the Actuating Layer embodies the system within the physical world.

The development process of a MAS is similar to that of any other software system. Modeling languages, software engineering techniques, and different tools can support it. Essentially, the designer programs the MAS using an existing BDI framework tailored to their specific requirements. However, developing an autonomous and cognitive system embedded in devices is a little more complex. Considering the state-of-art, we notice five steps (S1-S5), as shown in Table 3, that a designer must overcome during the development process. Each of these steps requires a specific skill that must be employed along with possible support tools. Moreover, the following steps can be approached either in a top-down or bottom-up way, since the *reasoning layer* is loosely coupled to the *controlling layer* once they exchange messages.

Table 4. System Requirements

Req.	Description
R1	*Remote MAS Management*. Allows uploading, starting, and stopping the MAS remotely, avoiding physical access to the device.
R2	*Connectivity*. Enables automatic wireless network connection and network switching without compromising reasoning.
R3	*Network Zone*. Allows the device to identify its network location and connected devices after reconnection or network changes.
R4	*Wi-Fi Direct*. Provides point-to-point communication between the device and a computer when no wireless infrastructure is available.
R5	*Remote Firmware Management*. Supports *Over the Air* (OTA) firmware updates without wired connections.
R6	*BDI Compliance*. Supports hosting a BDI-based MAS as the reasoning mechanism.
R7	*Remote MAS Debugging*. Enables remote monitoring of agents' mental states, such as beliefs and intentions.
R8	*Hardware Interface*. Allows the MAS to interact with the physical environment through microcontroller management.
R9	*Openness*. Supports MAS openness, including agent mobility and communication with other MAS.

As S1 concerns mounting physical components, no automated tool supports it. Some tools can facilitate the process by offering integrated and automated solutions to help the designer in S2-S4, as presented by Jesus et al. [43]. Nevertheless, no solution was found to help the designer in S5. About the fifth step, we notice that almost all works have used Raspberry Pi or BeagleBone to host the MAS. This choice is justified because these *High-End IoT Devices* have the resources to support embedded technologies for AIoT [3]. For this type of device, there are *High-End IoT OS* options, which rely on different GNU/Linux-based distributions and even an IoT-oriented version of Windows® [2,17].

However, performance remains a challenge in embedded cognitive systems. Even when the hardware platform is capable, the responsiveness of the agent depends on the cumulative overhead introduced by perception processing, plan selection, inter-layer communication, and OS-level services. As demonstrated by Gavigan and Esfandiari [15], poorly structured BDI agents or tightly coupled reasoning mechanisms can significantly increase the duration of the reasoning cycle, reducing the system's ability to react to stimuli in time-sensitive contexts. Additionally, Mohajeri Parizi et al. [28] show that the execution architecture of agent frameworks—whether interpreter-based, compiled, or actor-based—substantially impacts concurrency, scalability, and throughput, reinforcing the need for lightweight reasoning pipelines in embedded scenarios. Likewise, experiments conducted by Amaral [1] indicate that CPU load, memory consumption, and latency become critical when these systems operate in AIoT scenarios.

Beyond performance constraints, as shown by Liu et al. [25], when using a general-purpose OS, default configurations are not adequate for embedded AI

systems—such as BDI-based MAS. According to Zhang and Huang [47], properly configuring a general-purpose OS is a complex task that demands domain knowledge and experience; for instance, the Linux kernel exposes more than 17 thousand configuration options for tuning. Consequently, adapting such systems to host embedded cognitive applications may significantly increase the complexity of the development process. This complexity also arises from the diversity of technologies involved and the breadth of knowledge required from the development team, since the development process spans five distinct steps—from assembling hardware resources to programming cognitive behavior using BDI abstractions.

Across these steps, we observe that any solution intended to support this workflow must address a set of desirable requirements related to hosting a BDI-based MAS in embedded devices. These requirements directly reflect the challenges faced during the development steps S1–S5, particularly those related to deployment, connectivity, and system configuration, which remain largely unsupported by existing MAS frameworks and middleware solutions. These requirements are listed in Table 4.

In the literature, efforts are concentrated primarily on providing middleware and frameworks that assist in steps 3 and 4, while the remaining steps—equally important for building autonomous and cognitive devices—remain largely unsupported. The retrieved technologies have indeed been used to develop embedded autonomous solutions with agents, and some have been extended to enable hardware interfacing and IoT communication. However, these works generally comply only partially with requirements R6–R8 and, in a few cases, R9—as none of them provides support for R1–R5 at the operating-system level.

For example, the use of ARGO [34], a Jason-extended architecture that interfaces microcontrollers, allowing BDI agents to access sensors and actuators by exchanging messages, has been used on different editions of the Raspberry Pi and on the Banana Pi, typically running Raspberry Pi OS, a general-purpose OS not dedicated to agent systems. With ARGO, the designer must still implement services to satisfy R1–R5, as the OS provides none of them. As reflected in Table 5, ARGO satisfies R6, R7, and R8, but does not provide openness (R9). To extend support for JaCaMo, ARGO was later repackaged as an importable module [24]. Similarly, the JasonEmbedded [33], in turn, significantly reduces this gap by integrating IoT communication, hardware interfaces, and MAS abstractions into a unified distribution derived from Jason. JasonEmbedded satisfies R6, R7, R8, and R9, although it still relies on Raspberry Pi OS and therefore lacks support for R1–R5.

Another relevant technology is the Robot Operating System (ROS) [37], a modular framework composed of tools, libraries, and algorithms organized as a communication layer running over a host operating system. Although widely used in robotics and compatible with MAS-based solutions, ROS supports S3, providing interfaces between robotic platforms and embedded software. It depends entirely on the host OS for services such as network management, debugging, and firmware updates, and thus satisfies none of the requirements

R1–R5. ROS does not provide native support for BDI agents, and consequently does not meet R6–R9. Nevertheless, works integrating ROS and Jason show that embedding BDI agents in robotic systems is feasible. For instance, JasonROS [42] and later applications [31] demonstrate that BDI reasoning can simplify robotic-agent development. These solutions satisfy R6–R8 but still lack agent mobility (R9) and do not address the OS-level requirements R1–R5.

As an alternative to Raspberry Pi OS, Windows 10 IoT is a Windows-based system prepared for IoT devices and offering built-in services that satisfy R1 and R3. Our research found no evidence of Windows IoT being used in BDI-based MAS, which is expected since the technology was not designed for agent systems. It does not support requirements R6–R9. However, due to its IoT-oriented features, Windows IoT remains a candidate worth exploring for intelligent autonomous applications using agents, provided that MAS mechanisms are implemented on top of it.

Despite the advances achieved by these technologies, the comparative analysis in Table 5 shows that none of the solutions simultaneously satisfies the performance demands and the development requirements (R1–R9). Current approaches either rely on general-purpose operating systems that hinder timely responsiveness, or depend on middleware and ad-hoc extensions that only partially address the complexity of hosting cognitive agents in constrained devices.

Table 5. Comparison of solutions to embedded BDI-based MAS.

Solution	R1	R2	R3	R4	R5	R6	R7	R8	R9
Windows 10 IoT	√	X	√	X	X	X	X	X	X
RPI	X	X	X	X	X	X	X	X	X
RPI + Jason	X	X	X	X	X	√	√	X	X
RPI + Jason + ARGO	X	X	X	X	X	√	√	√	X
RPI + JaCaMo	X	X	X	X	X	√	√	X	X
RPI + JaCaMo + ARGO	X	X	X	X	X	√	√	√	X
RPI + JasonEmbedded	X	X	X	X	X	√	√	√	√
RPI + Arduino	X	X	X	X	√	X	X	X	X
RPI + Arduino + JasonEmbedded	X	X	X	X	√	√	√	√	√
ROS	X	X	X	X	X	X	X	X	X
ROS + JasonROS	X	X	X	X	X	√	√	√	X

This reveals a clear gap: the absence of an operating-system level infrastructure specifically designed to support the deployment, management, and operation of embedded BDI-based multi-agent systems. As a consequence, designers are required to manually configure networking, firmware deployment, remote management, and debugging services, which significantly increases the complexity of developing and maintaining embedded cognitive devices.

3 A Specific-Purpose Operating System to Support Embedded BDI-Based MAS

The development of embedded BDI-based MAS involves challenges that extend beyond the design of the agents themselves. Designers must deal with heterogeneous hardware platforms, microcontroller programming, network configuration, and the deployment and monitoring of agents. In addition, executing such systems requires managing system-level services including connectivity, firmware updates, and debugging support, which significantly increase the complexity of the development process. In practice, most current solutions rely on general-purpose operating systems to host the MAS. However, these systems were not designed for BDI-agent-based applications. Consequently, several functionalities required for deploying these systems must be implemented manually, typically through ad-hoc scripts or external tools. This increases engineering effort, raises the likelihood of configuration errors, and makes deployments hard.

The mapping shows that solutions only partially address the requirements to support this class of systems. While some frameworks facilitate agent reasoning or hardware interfacing, other capabilities such as remote MAS management, automatic network configuration, firmware deployment, and debugging support remain unsupported at the operating-system level. Besides, from a design perspective, supporting autonomous embedded systems requires an infrastructure capable of handling device management, network connectivity, firmware deployment, and monitoring of agents while reducing the configuration burden placed on system designers. Considering these requirements, we extend the observed MAS architecture by introducing a specialized operating-system-level layer capable for managing system services required for the deployment and operation.

3.1 An Extension of the Embedded BDI-Based MAS Architecture

The architecture of embedded BDI-based MAS traditionally comprises four layers that connect hardware components to the reasoning system. In this work, we introduce an additional layer dedicated to system and network management. The resulting *ACIRMa* architecture therefore consists of five layers, as illustrated in Fig. 2, and is formally defined as:

$$ACIRMa = (D, F, P, L, E)$$

where:

D is the set of operating-system daemon services, where: $D = (d_1, d_2, ..., d_n)$;
F is a framework accountable for executing the cognitive layer (e.g., Jason, JaCaMo, JasonEmbedded);
P is a communication protocol or a middleware that enables interaction between the agents and the physical devices (e.g., Javino, ROS, BrickPi);
L is the set of low-end IoT devices, where $L = (l_1, l_2, ..., l_n)$;
E is the set of sensors or effectors, where: $E = (e_1, e_2, ..., e_n)$.

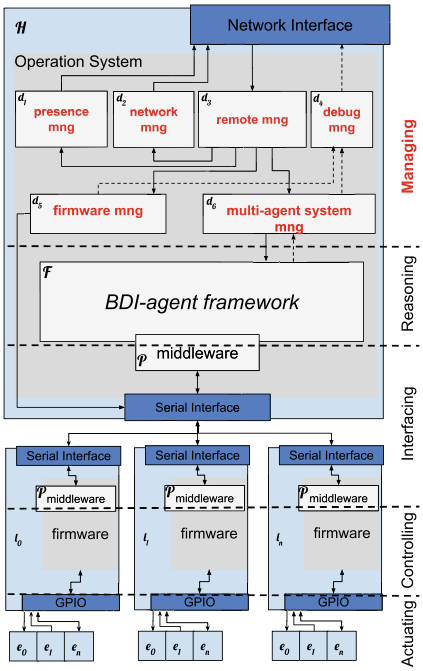


Fig. 2. ACIRMa architecture for embedded BDI-based MAS.

The lowest layer is the *Actuating Layer (L1)*, which includes the physical sensors and effectors (E_i) situated in the real-world environment. Each low-end device $l_i \in L$ is associated with a subset $E_i \subseteq E$ representing the sensors and/or effectors connected to that device, with the constraint that $E_i \neq \emptyset$, ensuring that every device (l_i) participates in either perception, actuation, or both. The next layer is the *Controlling Layer (L2)*, composed of the low-end IoT devices ($l_i \in L$, where $|L| \geq 1$) accountable for interacting with the physical components through firmware execution. Above it, the *Interfacing Layer (L3)* enables communication between the high-end IoT device (H) and the external hardware components. This layer relies on the protocol P to support firmware interaction with the connected L_i devices and to mediate the communication between the reasoning layer and the physical body of the system. The *Reasoning Layer (L4)* hosts the agent framework (F), accountable for executing the cognitive behavior of the system and managing the agents' beliefs, goals, and intentions.

Finally, the proposed *Managing Layer (L5)* provides management services (D) accountable for system configuration, network management, device administration, logging, and MAS lifecycle management. By integrating these capabilities at the operating-system level, the architecture simplifies the deployment and operation of embedded BDI-based MAS. To address the requirements R1–R9 identified in the previous section, the Managing Layer provides six operating-system services: d_1 a presence manager to solve R3; d_2 a network manager to

solve R2 and R4; d_3 a remote management to solve R1; d_4 a debug manager to solve R7; d_5 a firmware management to solve R5; d_6 a MAS manager to solve R6, R8, and R9.

It is important to emphasize that the novelty of the proposed approach does not lie in the individual management services themselves, since functionalities such as remote configuration, logging, or firmware deployment are already available in general IoT ecosystems. Instead, the contribution lies in integrating these capabilities as a unified operating-system-level infrastructure explicitly designed around the deployment, execution, management, and debugging requirements of embedded BDI-based systems identified in the literature review.

3.2 ChonOS Implementation

The proposed architecture is instantiated through the *Cognitive Hardware on Network - Operational System* (chonOS), a GNU/Linux distribution based on Debian. Debian was selected because it is widely used as the basis for other Linux systems, distributed under free software licensing, and supports several hardware architectures including ARM. It provides a stable software environment commonly adopted by single-board computers such as Raspberry Pi.

Table 6. Main services implemented in chonOS.

d_1 **chonos-ddns**: implements a *Dynamic DNS* mechanism accountable for maintaining the embedded system's *FQDN* updated, allowing access through a domain name without manually identifying its IP address when networks change. This module also implements a *Simple Service Discovery Protocol* server to announce the device within the local network, enabling multi-agent systems to exchange information about available services.
d_2 **chonos-network**: implements automatic management for *IEEE 802.3* and *IEEE 802.11* network interfaces, allowing direct access to the management layer through wired or wireless connections without manual configuration of *DHCP* . The module also supports *Wi-Fi Direct* and link monitoring with automatic reconnection using *ICMP* , which is useful for mobile robots operating outside wireless network range
d_3 **chonos-sysconfig**: provides a web interface for remote system management, extending the chonIDE [43], allowing configuration of device identification, network parameters, and interaction with the reasoning and firmware layers without requiring physical peripherals such as keyboard, mouse, or monitor.
d_4 **chonos-log**: provides remote monitoring through a web interface, allowing inspection of agent output messages and monitoring of agent behavior.
d_5 **chonos-firmware**: enables remote deployment and updates of firmware for Arduino boards without disassembling the cyber-physical system, using a firmware OTA mechanisms.
d_6 **chonos-embeddedMAS**: supports the deployment and execution of BDI-based MAS in the reasoning layer, enabling designers to upload, start, and stop projects developed using the JasonEmbedded [33] framework.

chonOS is ready-to-use, designed to support the deployment of embedded BDI-based MAS. It integrates several existing technologies and introduces specialized services to abstract technological dependencies involved in embedding MAS into hardware platforms. In addition, it integrates a framework for Embedded MAS programming, enabling agents to control hardware devices and supporting the development of mobile or IoT-based autonomous systems.

Since the system is designed specifically to support BDI-based MAS, minimizing operating-system overhead is essential to maximize the memory and processing resources available to the reasoning layer. The L5 was implemented in chonOS as a set of modular system services that together form its remote management infrastructure. These services support device configuration, MAS deployment, firmware management, and system monitoring. The main system services currently implemented in chonOS are summarized in Table 6.

4 Evaluation

According to Peffers et al. [36], DSR artifacts must be evaluated in relation to the requirements of their application context, providing evidence that the artifact effectively addresses the identified problem. To this end, we evaluate the instantiation of the proposed model from two perspectives. The first, *design-time evaluation*, compares the development and deployment process, while the second, *run-time evaluation*, analyzes resource consumption during execution. We aim to determine whether each approach (specific or general OS) simplifies the development and deployment of embedded BDI-based systems and whether it introduces additional computational overhead at runtime.

We consider the UGV domain by implementing an obstacle-avoidance prototype. The experiment compares the Cognitive Hardware on Network - Operating System (chonOS) (specific-purpose) and Raspberry Pi OS (rpiOS) (general-purpose). The latter was selected because the literature shows that most implementations rely on Raspberry Pi devices. The same hardware prototype, firmware, and BDI-agent code were used in both experiments in order to isolate the effect of the operating system.

The experimental prototype consists of an educational robotic platform, the chonBot, a low-cost 2WD prototype designed for robotic agents. The platform belongs to a family of standardized robotic prototypes composed of a single-board computer, a prototyping microcontroller, and a set of sensors and actuators, allowing BDI agents to interact with the real-world environment [23]. The hardware and software components of the prototype were organized as: L4 is hosted on a Raspberry Pi Zero W (RPi0W) (1 GHz single-core CPU, 512 MB RAM, and IEEE 802.11 b/g/n wireless LAN); L3 employs the Javino middleware; L2 is hosted on an Arduino Uno; and L1 includes an ultrasonic distance sensor, two reflective optical sensors, a light-dependent resistor, six light-emitting diodes, a buzzer, and the two-wheel mobile robot platform.

4.1 Evaluation 1: Development Complexity

This *design-time evaluation* analyzes how the chosen approach influences the development process. Specifically, we compare the tasks required to prepare the operating system, deploy the microcontroller firmware, and deploy the BDI-based reasoning system. These activities correspond to the S2–S5 stages of development described in the literature. We consider three aspects: the number of required configuration steps, the need for physical access to the device, and the degree of manual configuration required from the designer.

The general-purpose baseline was intentionally evaluated using the standard Raspberry Pi OS environment instead of lightweight command-line-only distributions. This choice reflects the deployment workflow commonly adopted in the related works, where designers rely on graphical interfaces, remote desktop access, external configuration tools, and manual system configuration to manage embedded MAS applications. Since one of the goals of this work is to reduce the operational complexity imposed on designers, the evaluation considers not only resource consumption, but also the practical deployment and management effort required during the development lifecycle of embedded BDI-based MAS.

Table 7 summarizes the tasks required to prepare the operating-system environment to host the embedded MAS (S5). While the specific-purpose approach centralizes configuration and deployment activities through a web-based management interface, the general-purpose approach requires physical access to the device and several manual configuration procedures, including network setup, installation of software dependencies, activation of remote-access services, and additional system configuration tasks. These differences reflect the operational complexity typically imposed on designers when deploying embedded BDI-based MAS using general-purpose operating systems.

Table 8 compares the process of deploying firmware to the microcontroller (S2-S3). With chonOS, firmware can be deployed remotely using the web interface. In contrast, the general-purpose approach requires the installation of additional tools (e.g., a VNC client) to allow the use of the GUI on the device.

Finally, Table 9 compares the tasks required to deploy and execute the MAS (S4). The specific-purpose platform provides a centralized environment for deployment and monitoring of the MAS. In the general-purpose platform, the developer must interact via VNC to execute the system and observe the output of agent messages, and must also use a web browser to visualize the agent mind.

The number of configuration steps corresponds to the individual actions performed by the designer to complete each stage described in Tables 7-9. Each step represents a single manual action required during the deployment process. Table 10 summarizes the main differences between both OS approaches.

Considering only the preparation of the operating-system environment, the general-purpose approach required 71 min, whereas the specific-purpose approach required only 16 min. This represents a reduction of approximately 77.5% in setup time. Together with the reduction from 16 to 5 manual configuration actions, these results indicate that the specialized OS substantially simplifies the

Table 7. Tasks required to prepare the OS environment.

OS Type	Configuration Tasks
specific-purpose	1) The designer powered on the prototype. The chonOS created a Wi-Fi Direct network named "EmbeddedMAS". The designer connected a computer to this network. 2) Using a web browser, the designer accessed the management interface (https://192.168.168.1:3270), assigned a *device name* and entered the wireless network credentials.
general-purpose	1) The designer needed to disassemble the prototype to connect a keyboard and a monitor to the device. After that the designer powered on the prototype. 2) The wireless network had to be manually configured. 3) SSH access had to be enabled to allow remote file transfer. 4) A VNC server had to be enabled to allow remote graphical access to the device. 5) The Java JDK had to be installed to support the BDI framework. 6) The Arduino IDE had to be installed to allow firmware deployment. 7) The Javino middleware and its dependencies had to be downloaded and installed. 8) The JasonEmbedded framework had to be downloaded from the project repository.

Table 8. Tasks required to deploy firmware to the ATmega microcontroller.

OS Type	Deployment Tasks
specific-purpose	1) The designer downloaded the source code from the repository to the computer. 2) Using the web management interface (*https://[deviceName].bot.chon.group:3270/*), the designer uploaded the project, compiled the firmware, and deployed it remotely.
general-purpose	1) The designer needed to connect a monitor to verify the IP address assigned to the device in order to access it remotely. 2) The designer accessed the device through SSH and downloaded the project repository. 3) The designer accessed the GUI using VNC to open the Arduino IDE. 4) The firmware was compiled and deployed to the microcontroller.

deployment preparation of embedded BDI-based MAS. Moreover, most tasks in the rpiOS required direct physical interaction with the device, while the chonOS allowed all operations to be executed remotely through the web-based manager.

Table 9. Tasks required to execute and monitor the MAS.

OS Type	Deployment Tasks
specific-purpose	1) Using the web management interface (*https://[deviceName].bot.chon.group:3270/*), the developer remotely starts and monitors the MAS project.
general-purpose	1) The designer needed to connect a monitor to verify the IP address assigned to the device in order to access it remotely. 2) The developer accessed the device using VNC to ensure that the MAS process would remain active even if the remote session was closed. 3) A terminal session was opened within the GUI to execute the MAS project. 4) On the computer, the designer used a browser to visualize the agent mind (*http://[deviceIPAddress]:3272/*).

Table 10. Summary of development tasks supported by each OS approach.

Task	specific-purpose	general-purpose
OS Installation	manual process	manual process
Network Configuration	remotely (web)	wired
MAS Runtime Dependencies	pre-installed	manual installation required
MAS Deployment	remotely (web)	remotely (VNC)
Debugging	remotely (web)	log terminal + remotely

An additional limitation observed in the general-purpose OS concerns the monitoring of agent minds. The JasonEmbedded framework exposes the internal reasoning logs through a web interface. However, due to the hardware constraints of the RPi0W (armv6l architecture), no browsers are available to run this interface. As a consequence, the debugging interface had to be accessed externally through the device IP address.

Finally, we notice that a specialized OS can reduce the configuration complexity required to deploy an embedded BDI-based MAS, positively answering RQ1.

4.2 Evaluation 2: Performance

The *run-time evaluation* analyzes the performance of the embedded BDI-based system during execution. Since such systems are typically deployed on resource-constrained devices, this experiment verifies whether the specialized OS introduces additional computational overhead during the execution of the BDI reasoning cycle. This evaluation focuses on analyzing the impact of the operating-system infrastructure on the execution of an embedded multi-agent system hosted locally on the device. Therefore, the experiment emphasizes the inter-

action between embedded hardware, inter-agent communication, and the BDI reasoning cycle during execution.

The experiment was executed using the same hardware platform described previously in order to isolate the effect of the operating system. Each benchmark was executed ten times for each OS. The reported values correspond to the average results obtained across these executions in order to mitigate possible variations caused by system scheduling and background processes. Both systems executed the same firmware and multi-agent program. To stress the reasoning cycle, we implemented a benchmark in which the Arduino microcontroller generated sequential numeric perceptions continuously during a 60-s interval.

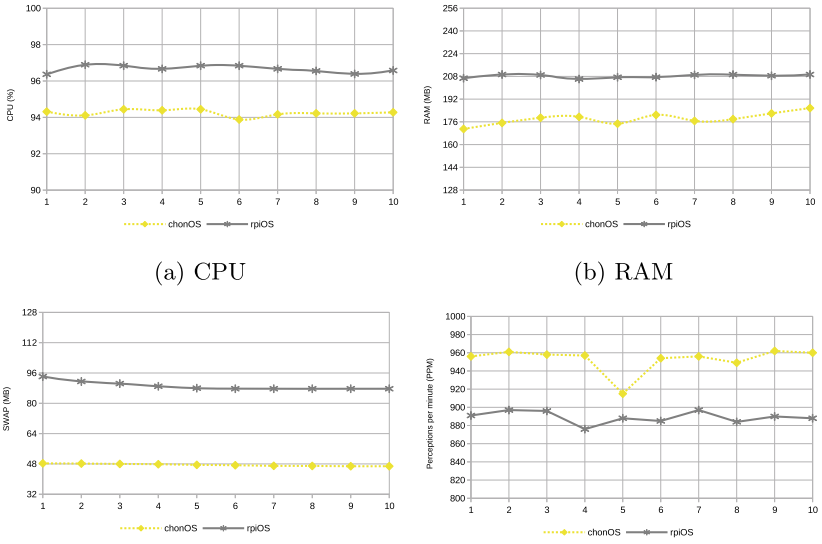
The benchmark employed two cooperative BDI agents within the MAS. The first agent was responsible for interacting with the embedded environment by receiving sequential numeric perceptions generated by the microcontroller. After receiving a perception, this agent forwarded the value through inter-agent communication to a second cognitive agent responsible for computing a Diffie–Hellman cryptographic key using modular exponentiation. Once the computation was completed, the result was returned to the percept-processing agent, allowing the next perception to be processed. This interaction created a continuous perception–reasoning cycle involving hardware interaction, inter-agent communication, and cognitive processing within the embedded MAS.

The total number of *processed perceptions during the one-minute interval* (PPM) was recorded as the system throughput. During execution, system resource consumption was monitored every five seconds, collecting CPU utilization, RAM usage, and swap activity. Figure 3 summarizes the measured results.

The experiment was executed ten times for each operating-system configuration. The reported values correspond to the mean and standard deviation of the observed measurements. The results indicate that both operating systems presented similar CPU utilization, with averages of $94.2 \pm 0.2\%$ for chonOS and $96.7 \pm 0.2\%$ for rpiOS. However, the specialized OS consumed less memory, averaging 178.2 ± 4.1 MB compared with 208.2 ± 1.2 MB in rpiOS, corresponding to a reduction of approximately 14.4%. Swap usage also presented a significant difference, with averages of 47.5 ± 0.6 MB in chonOS and 89.2 ± 2.2 MB in rpiOS, representing a reduction of approximately 46.8%. Regarding throughput, the MAS processed an average of 952.8 ± 13.8 PPM in chonOS and 889.2 ± 6.0 PPM in rpiOS, corresponding to an improvement of approximately 7.2%.

Overall, the results indicate that the specialized operating system does not introduce additional computational overhead when compared with the general-purpose environment. On the contrary, the reduction in memory and swap usage allows slightly higher processing throughput for the embedded MAS.

We notice that the use of a specialized OS does not negatively affect runtime performance and may even improve system efficiency, thus addressing RQ2.



(a) CPU (b) RAM

(c) SWAP (d) PPM

Fig. 3. Benchmark results comparing system resource usage during the execution of the embedded MAS.

5 Reproducibility

To facilitate replication of the research and provide transparency regarding the developed artifact and the evaluation process, we provide a dedicated page[1] containing detailed information about the experiments, source code, benchmark datasets, installation instructions for chonOS, videos demonstrating the prototype execution, and the data generated during the snowballing process.

6 Conclusion

The development of embedded BDI-based systems involves challenges that extend beyond agent programming. Designers must deal with heterogeneous hardware platforms, firmware development, network configuration, and system-level services required to deploy and maintain autonomous devices. As discussed in the literature, existing approaches rely primarily on general-purpose operating systems combined with middleware or framework extensions, which leaves several operational concerns unsupported and increases the complexity of the development process.

This work addressed this gap by proposing an operating-system-level infrastructure specifically designed to support this class of systems. Following the DSR

[1] https://papers.chon.group/ICCSA/chonOS/.

methodology, we identified the limitations of current solutions, derived a set of requirements (R1–R9), and designed an architectural model that extends the traditional embedded MAS architecture with a dedicated management layer. This model was instantiated through the implementation of *chonOS*, a GNU/Linux-based distribution that integrates services for network management, remote deployment, firmware updates, MAS lifecycle management, and system monitoring.

The evaluation considered both *design-time* and *run-time* perspectives. The first evaluation showed that the specialized operating system substantially reduces the configuration effort required to deploy embedded BDI-based MAS, decreasing the number of manual configuration actions from sixteen to five and reducing the environment preparation time by approximately 77.5%. This result indicates that integrating management services directly into the OS simplifies the development and deployment workflow. The second evaluation demonstrated that this specialization does not introduce additional computational overhead. On the contrary, the experimental results showed lower memory and swap consumption when compared with a general-purpose OS, while maintaining similar CPU utilization and achieving a slightly higher perception-processing throughput. These results provide evidence that the proposed artifact effectively supports the operational requirements of embedded MAS while preserving runtime efficiency.

The contribution of this work lies in the design of the ACIRMa model, an architectural extension that introduces a management layer for embedded BDI-based systems, and in its instantiation through chonOS, a specialized operating-system distribution that implements the proposed services. By integrating system management and cognitive agent execution within a unified infrastructure, the proposed model and its implementation reduce configuration complexity and support key operational requirements for embedded MAS. As a result, this work contributes to making the development and deployment of embedded BDI-based systems more accessible, reproducible, and manageable in practical scenarios.

Future work includes extending chonOS to support additional agent frameworks, exploring lightweight kernel configurations to further reduce system overhead, and evaluating the platform in larger multi-device deployments involving cooperative autonomous agents.

References

1. Amaral, C.J.: Embedding Multi-Agent System Frameworks (2018). https://periodicos.univali.br/index.php/acotb/article/view/12867
2. Antony, A., S., S.: A review on IoT operating systems. IJCA **176**(24) (2020). https://doi.org/10.5120/ijca2020920245
3. Bansal, S., Kumar, D.: IoT ecosystem: a survey on devices, gateways, operating systems, middleware and communication. Int. J. Wireless Inf. Netw. **27**(3), 340–364 (2020). https://doi.org/10.1007/s10776-020-00483-7
4. Boissier, O., Bordini, R.H., Hübner, J.F., Ricci, A., Santi, A.: Multi-agent oriented programming with JaCaMo. Sci. Comput. Programm. **78**(6) (2013). https://doi.org/10.1016/j.scico.2011.10.004

5. Bordini, R.H., El Fallah Seghrouchni, A., Hindriks, K., Logan, B., Ricci, A.: Agent programming in the cognitive era. Auton. Agent. Multi-Agent Syst. **34**(2), 1–31 (2020). https://doi.org/10.1007/s10458-020-09453-y
6. Brandão, F.C., Lima, M.A.T., Pantoja, C.E., Zahn, J., Viterbo, J.: Engineering approaches for programming agent-based IoT objects using the resource management architecture. Sensors **21**(23) (2021). https://doi.org/10.3390/s21238110
7. Bratman, M.E., Israel, D.J., Pollack, M.E.: Plans and resource-bounded practical reasoning. Comput. Intell. **4**(3), 349–355 (1988). https://doi.org/10.1111/j.1467-8640.1988.tb00284.x
8. Burguillo, J.C.: Multi-agent Systems. In: Self-organizing Coalitions for Managing Complexity (2018). https://doi.org/10.1007/978-3-319-69898-4_5
9. BBădică, C., Budimac, Z., Burkhard, H.D., Ivanovic, M.: Software agents: Languages, tools, platforms. Comput. Sci. Inform. Syst. **8**(2) (2011) https://doi.org/10.2298/CSIS110214013B
10. Castro, L.F.S.d., Manoel, F.C.P.B., Souza de Jesus, V., Pantoja, C.E., Pinz Borges, A., Vaz Alves, G.: Integrating embedded multiagent systems with urban simulation tools and IoT applications. RITA **29**(1), 81–90 (2022). https://doi.org/10.22456/2175-2745.110837
11. De Silva, L., Meneguzzi, F., Logan, B.: BDI agent architectures: a survey. In: IJCAI'20 (2020). https://dl.acm.org/doi/abs/10.5555/3491440.3492124
12. D'Inverno, M., Luck, M.: Engineering AgentSpeak(L): a formal computational model. J. Log. Comput. **8**(3), 233–260 (1998). https://doi.org/10.1093/logcom/8.3.233
13. Dresch, A., Lacerda, D.P., Antunes, J.A.V.: Proposal for the conduct of design science research. In: Design Science Research, pp. 117–127 (2015). https://doi.org/10.1007/978-3-319-07374-3_6
14. Gavigan, P., Esfandiari, B.: Agent in a box: a framework for autonomous mobile robots with beliefs, desires, and intentions. Electronics **10**(17), 2136 (2021). https://doi.org/10.3390/electronics10172136
15. Gavigan, P., Esfandiari, B.: Quantifying the relationship between software design principles and performance in Jason: a case study with simulated mobile robots. Ann. Math. Artif. Intell. **92**(4), 775–795 (2024). https://doi.org/10.1007/s10472-023-09844-3
16. Hou, K.M., Diao, X., Shi, H., Ding, H., Zhou, H., De Vaulx, C.: Trends and challenges in AIoT/IIoT/IoT implementation. Sensors **23**(11), 5074 (2023). https://doi.org/10.3390/s23115074
17. Javed, F., Afzal, M.K., Sharif, M., Kim, B.S.: Internet of Things (IoT) operating systems support, networking technologies, applications, and challenges: a comparative review. IEEE Commun. Surv. Tutor. **20**(3) (2018). https://doi.org/10.1109/COMST.2018.2817685
18. Jesus, V.S.D., Lazarin, N.M., Pantoja, C.E., Manoel, F.C.P.B., Alves, G.V., Viterbo, J.: A middleware for providing communicability to Embedded MAS based on the lack of connectivity. Artif. Intell. Rev. **56**(S3), 2971–3001 (2023). https://doi.org/10.1007/s10462-023-10596-z
19. Jesus, V.S.d., Pantoja, C., Manoel, F., Alves, G., Viterbo, J., Bezerra, E.: Bio-Inspired protocols for embodied multi-agent systems. In: 13th ICAART (2021). https://doi.org/10.5220/0010257803120320
20. Karaduman, B., Tezel, B.T., Challenger, M.: Enhancing BDI agents using fuzzy logic for CPS and IoT interoperability using the JaCa platform. Symmetry **14**(7), 1447 (2022). https://doi.org/10.3390/sym14071447

21. Karaduman, B., Tezel, B.T., Challenger, M.: Rational software agents with the BDI reasoning model for Cyber–Physical Systems. Eng. Appl. Artif. Intell. **123** (2023). https://doi.org/10.1016/j.engappai.2023.106478
22. Karaduman, B., Tezel, B.T., Challenger, M.: On the impact of fuzzy-logic based BDI agent model for cyber–physical systems. Expert Syst. Appl. **238**, 122265 (2024). https://doi.org/10.1016/j.eswa.2023.122265
23. Lazarin, N.M., Pantoja, C.E., Viterbo, J.: Towards a toolkit for teaching AI supported by robotic-agents: proposal and first impressions. In: WEI 2023 (2023). https://doi.org/10.5753/wei.2023.229753
24. Lazarin, N.M., Pantoja, C.E., Viterbo, J.: Dealing with the unpredictability of physical resources in real-world multi-agent systems. In: ICAART 2023, vol. 14546. Cham (2024). https://doi.org/10.1007/978-3-031-55326-4_3
25. Liu, X., et al.: Operating systems for resource-adaptive intelligent software: challenges and opportunities. ACM Trans. Internet Technol. **21**(2) (2021). https://doi.org/10.1145/3425866
26. Mascardi, V., et. al: Engineering multi-agent systems: state of affairs and the road ahead. ACM SIGSOFT Softw. Eng. Notes **44**(1), 18–28 (Mar 2019). https://doi.org/10.1145/3310013.3322175
27. Menegol, M.S., Hübner, J.F., Becker, L.B.: Coordinated UAV search and rescue application with JaCaMo. In: PAAMS 2018, vol. 10978, pp. 335–338 (2018). https://doi.org/10.1007/978-3-319-94580-4_33
28. Mohajeri Parizi, M., Sileno, G., Van Engers, T., Klous, S.: Run, agent, run! architecture and benchmarking of actor-based agents. In: AGERE 2020. ACM (2020). https://doi.org/10.1145/3427760.3428339
29. Morais, M.G., Meneguzzi, F.R., Bordini, R.H., Amory, A.M.: Distributed fault diagnosis for multiple mobile robots using an agent programming language. In: ICAR 2015. IEEE (2015). https://doi.org/10.1109/ICAR.2015.7251486
30. Ojo, M.O., Giordano, S., Procissi, G., Seitanidis, I.N.: A review of low-end, middle-end, and high-end Iot devices. IEEE Access **6** (2018). https://doi.org/10.1109/ACCESS.2018.2879615
31. Onyedinma, C., Gavigan, P., Esfandiari, B.: Toward campus mail delivery using BDI. J. Sens. Actuator Netw. **9**(4), 56 (2020). https://doi.org/10.3390/jsan9040056
32. Pantoja, C., Soares, H.D., Viterbo, J., Seghrouchni, A.E.F.: An Architecture for the Development of Ambient Intelligence Systems Managed by Embedded Agents, pp. 215–249 (July 2018). https://doi.org/10.18293/SEKE2018-110
33. Pantoja, C.E., Jesus, V.S.D., Lazarin, N.M., Viterbo, J.: A spin-off version of Jason for Iot and embedded multi-agent systems. In: BRACIS 2023, vol. 14195. Cham (2023). https://doi.org/10.1007/978-3-031-45368-7_25
34. Pantoja, C.E., Stabile, M.F., Lazarin, N.M., Sichman, J.S.: ARGO: An extended jason architecture that facilitates embedded robotic agents programming. In: EMAS 2016 (2016). https://doi.org/10.1007/978-3-319-50983-9_8
35. Pantoja, C.E., Viterbo, J., Seghrouchni, A.E.F.: From thing to smart thing: towards an architecture for agent-based AmI Systems. In: KES-AMSTA 2019 (2020). https://doi.org/10.1007/978-981-13-8679-4_5
36. Peffers, K., Rothenberger, M., Tuunanen, T., Vaezi, R.: Design science research evaluation. In: DESRIST 2012. Springer (2012). https://doi.org/10.1007/978-3-642-29863-9_29
37. Quigley, M., et al.: ROS: an open-source robot operating system. In: ICRA 2009 (2009). https://scholar.google.com/scholar?cluster=143767492575573826

38. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: Van de Velde, W., Perram, J.W. (eds.) MAAMAW 1996. LNCS, vol. 1038, pp. 42–55. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0031845
39. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach. Always learning, Pearson, $3T^{\underline{O}}$ edn. (2016)
40. Santos, F., Hübner, J., Becker, L.: Design and analysis of a BDI agent model for route planning in a UAV. In: WESAAC 2015. SBC (2015). https://doi.org/10.5753/wesaac.2015.33313
41. Sichman, J.S., Demazeau, Y., Boissier, O.: When can knowledge-based systems be called agents? In: IX SBIA, vol. 9, pp. 172–185. SBC, Rio de Janeiro (1992). https://scholar.google.com.br/scholar?cluster=12989357293789094729
42. Silva, G.R., Becker, L.B., Hübner, J.F.: Embedded architecture composed of cognitive agents and ROS for programming intelligent robots. IFAC-PapersOnLine **53**(2), 10000–10005 (2020). https://doi.org/10.1016/j.ifacol.2020.12.2718
43. Souza De Jesus, V., Mori Lazarin, N., Pantoja, C.E., Vaz Alves, G., Ramos Alves De Lima, G., Viterbo, J.: An IDE to support the development of embedded multiagent systems. In: PAAMS 2023. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-37616-0_29
44. Sycara, K.P.: Multiagent systems. AI Mag. **19**(2), 79–79 (1998). https://doi.org/10.1609/aimag.v19i2.1370
45. Wohlin, C.: Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: EASE '14, pp. 1–10. ACM, UK (2014). https://doi.org/10.1145/2601248.2601268
46. Wooldridge, M., Jennings, N.R.: Intelligent agents: theory and practice. Knowl. Eng. Rev.**10**(2) (1995). https://doi.org/10.1017/S0269888900008122
47. Zhang, Y., Huang, Y.: "Learned": Operating Systems. ACM SIGOPS Oper. Syst. Rev. **53**(1) (2019). https://doi.org/10.1145/3352020.3352027